Performance

Introduction

In the Play+ ecosystem, performance is not a feature to be added later—it's a foundational commitment. We believe great design must feel fast. This helper is based on the concept of Performance by Default, where speed, responsiveness, and stability are architected and enforced from day one.

A performant application builds user trust and delight. This directly supports our core design pillars by creating an Engaging experience that feels fluid and responsive, and an Adaptive interface that performs well across all devices and network conditions. By ensuring a fast, stable experience, we also make our products more Intuitive and less frustrating to use.

Package Info

The Play+ performance helper and its associated CI configurations are included by default in the Golden Path starter kit.

Description

Package / Path	Golden Path (Recommended)
Pre-installed	/system/play.perf.ts
Uplift Path	npm install @playplus/perf

Folder Reference

The performance helper and its configuration follow our standardized folder structure for core system logic.

File / Directory	Purpose & Guidelines
system/play.perf.ts	The core performance helper. It provides utilities for deferring tasks and monitoring real-world performance.

File / Directory	Purpose & Guidelines
config/play.performance.config.json	User-overridable configuration for performance budgets (Lighthouse) and monitoring settings.
reports/performance/	The git-ignored directory where Lighthouse CI reports are saved for local inspection.

Helper - Pillars Alignment

The playperf helper is a direct implementation of our core design pillars, focused on the user's perception of speed.

Pillar	How This Helper Aligns
Engaging	Primary Pillar: Ensures snappy interactions and fluid animations, which are key to a delightful and engaging user experience.
Adaptive	Enforces performance budgets that ensure the application is fast and responsive on a wide range of devices and networks.
Intuitive	A fast and stable application feels more predictable and intuitive, as users aren't left waiting or dealing with layout shifts.

Helper Overview

The playperf toolchain is a combination of automated guardrails and developer-facing utilities designed to abstract the plumbing of performance management. Instead of manually configuring performance budgets or writing complex code to defer non-critical tasks, developers can rely on the Play+ system to handle it.

It automates performance quality control in two key ways:

- Preventative Guardrails : In the CI/CD pipeline, playperf automatically runs Lighthouse audits against every pull request. If key metrics (like LCP or CLS) exceed the configured budgets, the build fails, preventing performance regressions from ever

reaching production.

• Developer Empowerment Utilities : It provides a simple runtime helper with methods like defer() and monitor() that make it trivial to implement common performance patterns without writing boilerplate code.

The goal is to make high performance the default path, not an extra chore.

Config Options

Global performance budgets and settings are managed in config/play.performance.config.json . These values are consumed by the CI pipeline and runtime helpers.

Config Key Table

Config Key	Default Value	Description	Recommended Value
lighthouse.lcp	2500	Lighthouse budget for Largest Contentful Paint (ms).	2500
lighthouse.cls	0.1	Lighthouse budget for Cumulative Layout Shift.	0.1
lighthouse.inp	200	Lighthouse budget for Interaction to Next Paint (ms).	200
lighthouse.maxBundleSize	250	Lighthouse budget for initial JS bundle size (KB, gzipped).	250
enforce.blockBuildOnFail	true	If true, the CI build will fail if performance budgets are exceeded.	true

Config Key	Default Value	Description	Recommended Value
enforce.enableBundleDiff	true	If true, PRs will be flagged with any unexpected changes to bundle size.	true
monitor.webVitals	true	If true, enables the real-user monitoring of Core Web Vitals via playperf.monitor.	true
monitor.sentry	true	Enables integration with Sentry Performance for transaction tracking.	true
assist.highlightHeavyComponents	true	In dev mode, flags components that are computationally expensive or re-rendering unnecessarily.	true

Helper Methods

Core Methods

Method Name	Description	Signature
defer	Schedules non-critical tasks to run during browser idle time, preventing them from blocking critical rendering.	defer(callback: () => void, options?: { priority: 'high' | 'low', timeout: number }): void

Method Name	Description	Signature
monitor	Initializes comprehensive performance monitoring including Core Web Vitals, long tasks, and bundle size tracking.	monitor(options: { onReport: (metric) => void, enableWebVitals?: boolean, enableLongTasks?: boolean, enableBundleMonitoring?: boolean }): void
measure	Measures the performance of synchronous functions.	measure<T>(name: string, fn: () => T): T
measureAsync	Measures the performance of asynchronous functions.	measureAsync<T>(name: string, fn: () => Promise<T>): Promise<T>

Angular Integration

PlayPerfService

Angular service wrapper that integrates with Play+ logging and provides component-specific performance utilities.

Performance Directive

Automatically monitors component render times and highlights heavy components.

Performance Pipe

Measures and reports template expression performance.

Usage Examples

React: Deferring a Non-Critical Script

This example shows how to defer the initialization of an analytics script so it doesn't interfere with the initial page load.

Angular: Monitoring Real-World Performance

This example shows how to initialize Web Vitals monitoring in your main application component and log the results.

Basic Usage Examples

Additional Info

Why We Created This Helper

Web performance is critical, but its tooling is complex. Without a dedicated system, each team would need to:

- Manually set up, configure, and maintain Lighthouse CI for every project.
- Write custom logic to track bundle sizes.
- Implement their own utilities for deferring scripts or monitoring Web Vitals.

This is inefficient and leads to inconsistent standards. The playperf helper automates the enforcement and simplifies the implementation of performance best practices. It provides a pre-configured safety net and easy-to-use tools so developers can build fast experiences by default.

Performance Budgets

All Play+ applications must meet these baseline performance standards:

Metric	Target	Budget
LCP (Largest Contentful Paint)	< 2.5s	2500ms
INP (Interaction to Next Paint)	< 200ms	200ms
CLS (Cumulative Layout Shift)	< 0.1	0.1
JS Bundle Size (gzipped)	< 250KB	250KB

Metric	Target	Budget
Main Thread Long Tasks (>50ms)	0	0

Best Practices

Component Optimization

Change Detection Strategy

Use OnPush change detection for components that don't need frequent updates:

TrackBy Functions

Always provide trackBy functions for ngFor loops:

Lazy Loading

Use lazy loading for routes and components:

Data Management

Caching Strategy

Use Play+ caching for expensive operations:

Virtual Scrolling

Use virtual scrolling for large lists:

Image Optimization

Lazy Loading

Always lazy load images:

Modern Formats

Use WebP with fallbacks:

Explicit Dimensions

Always specify width and height to prevent layout shifts:

Forbidden Patterns

Blocking Operations

Synchronous API Calls

Expensive Template Expressions

Manual setTimeout/setInterval

Required Patterns

Performance Monitoring

Component Performance Tracking

Async Data Loading

Testing Performance

Unit Testing

Integration Testing

Monitoring and Analytics

Core Web Vitals

- LCP : Largest Contentful Paint - measures loading performance
- FID : First Input Delay - measures interactivity
- CLS : Cumulative Layout Shift - measures visual stability

Custom Metrics

- Component render times
- API response times
- Bundle size changes
- Long task detection

Reporting

Performance metrics are automatically logged and can be sent to:

- Play+ logging system
- Analytics platforms
- Monitoring dashboards
- CI/CD pipelines

Developer Checklist

- Are all non-critical third-party scripts or tasks wrapped in playperf.defer() ?
- Is the code for my route being dynamically imported (code-splitting)?
- Are all images lazy-loaded and served in modern formats (e.g., WebP) with explicit width and height attributes to prevent CLS?
- Have I used skeleton loaders for data-heavy components instead of spinners?
- For long lists, am I using a virtualized scrolling solution?
- Are pure components memoized using React.memo or Angular's OnPush change detection strategy?
- Have I analyzed the bundle size impact of any new dependencies I've added?
- Does my feature pass the Lighthouse CI performance budget checks?

Performance Checklist

Development

- All non-critical tasks use playperf.defer()
- Components use OnPush change detection where appropriate
- All ngFor loops have trackBy functions
- Images are lazy loaded with explicit dimensions
- Routes are lazy loaded
- Heavy computations are measured with playperf.measure()

Testing

- Performance monitoring is initialized in app component
- Component render times are within 16ms budget
- Bundle size is under 250KB limit
- Lighthouse scores meet performance budgets
- Long tasks are eliminated

Production

- Core Web Vitals are monitored
- Performance metrics are logged
- Bundle analysis is performed
- CDN is configured for static assets
- Gzip compression is enabled

Recommended Monitoring Tools

For a full view of real-world performance, we recommend integrating with:

- Web Vitals JS Library : To send Core Web Vitals from the browser to your analytics or logging provider.
- Sentry Performance : To track slow transactions and identify main thread stalls.
- SpeedCurve or New Relic : For in-depth Real User Monitoring (RUM) analysis over time, across geographies and devices.

Minimum Enforcement Thresholds

All Play+ applications in the Golden Path are held to these baseline standards, which are enforced automatically in the CI/CD pipeline.

Metric	Target
LCP (Largest Contentful Paint)	< 2.5s
INP (Interaction to Next Paint)	< 200ms
CLS (Cumulative Layout Shift)	< 0.1
JS Bundle Size (gzipped)	< 250KB
Main Thread Long Tasks (>50ms)	0